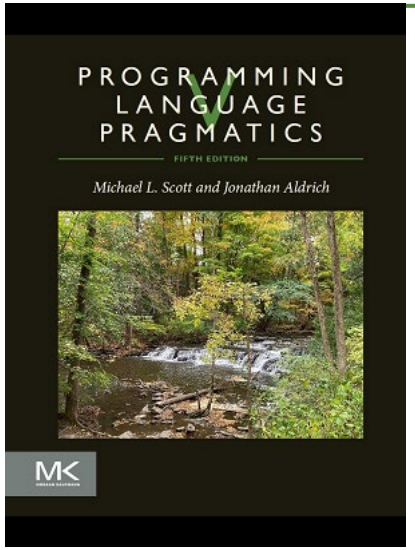
Chapter 6: Control Flow

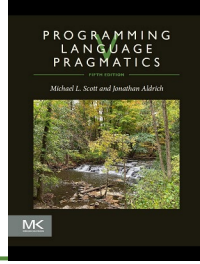


Programming Language Pragmatics, Fifth Edition

Michael L. Scott and Jonathan Aldrich

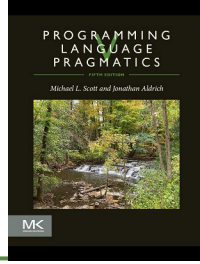
Copyright © 2024 by Jonathan Aldrich and Michael L. Scott

How to compile programs with control flow?



- How to represent Booleans
- Relevant x86-64 architecture & instructions
- How to compile:
  - If statements
  - Relational operators
  - Loops

Representing Booleans



- C: no Boolean type
  - 0 represents false
  - any other integer represents true
- More strongly-typed languages (Java, Rust, ML, …)
  - Boolean is a distinguished type
  - common choice: 0 for false, 1 for true: similar to C
- Another choice: *tagging*. Example (for 32 bits):
  - numbers represented by 31 bits, shifted left one bit
    - so 5 would be 0000 0000 0000 0000 0000 0000 0000 1010

5

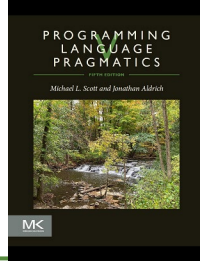
number tag

  - low bit of 1 means a Boolean, the 2nd lowest bit which Boolean
    - so true would be 0000 0000 0000 0000 0000 0000 0000 0011
  - can test type at run time in dynamic languages

true

boolean tag

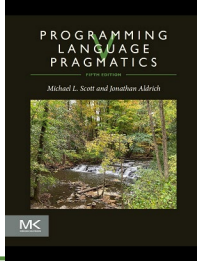
x86 control flow architecture



- `cmp v1, v2` computes `v1-v2` and sets *condition codes*
  - Kind of like `sub`, but does not change `v1`
    - Note: `sub` does not change the condition codes, but `subs` does
- Here are the condition codes
  - ZF: zero flag – set to 1 if v1 = v2
  - SF: sign flag – set to 1 if result is negative
  - OF: overflow flag – set to 1 when there is signed overflow (127+1 = -128)
  - CF: carry flag – set to 1 when there is unsigned overflow (255+1 = 0)
- Generally easier to think in terms of greater than, equal, etc.
  - See mnemonics on the next slide

*Note we are using NASM-style Intel syntax in this class, where destinations come first

x86 control flow architecture

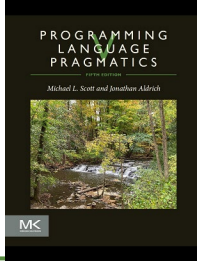


- `cmp v1, v2` computes `v1-v2` and sets *condition codes*
  - (without changing `v1`)
- These can be used by *conditional instructions*. Examples:

```
jmp label          ; transfers control to label, no matter what
je label           ; transfers control to label if equal (ZF = 1)
jne label          ; transfers control to label if not equal (ZF = 0)
jg label           ; transfers control to label if greater than (signed)
jge label          ; transfers control to label if greater than or equal (signed)
ja label           ; transfers control to label if above (unsigned)
jo label           ; transfers control to label if overflow (signed)
cmove v1, v2       ; moves v2 to v1 if equal
...
```

*Note we are using NASM-style Intel syntax in this class, where destinations come first

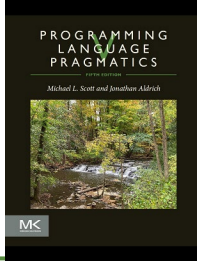
How to compile if statements



expr ::= if <cond> then <expr1> else <expr2>

```
    {cond_instrs}   ; leaves result in rax
    cmp rax, 0
    je else
    {then_instrs}   ; leaves result in rax
    jmp end
else:
    {else_instrs}   ; leaves result in rax
end:
```

What if we have nested if expressions?



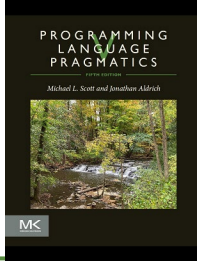

```
if c1 then if c2 then e1 else e2 else e3
```

➔

```
   {C1}
   cmp rax, 0
   je else
   {C2}
   cmp rax, 0
   je else
   {e1}
   jmp end
else:
   {e2}
end:
   jmp end
else:
   {e3}
end:
```

What's the problem here?

How to compile if statements (fixed)

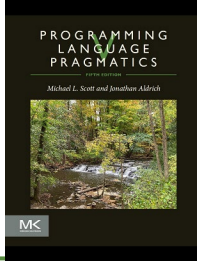


expr ::= if <cond> then <expr1> else <expr2>

```
    {cond_instrs}   ; leaves result in rax
    cmp rax, 0
    je else0
    {then_instrs}   ; leaves result in rax
    jmp end1
else0:
    {else_instrs}   ; leaves result in rax
end1:
```

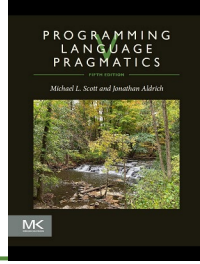
Add an index to make every label unique

Fixed version - nested if expressions



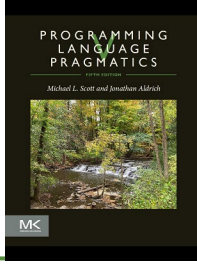

```
if c1 then if c2 then e1 else e2 else e3
➔
   {C1}
   cmp rax, 0
   je else0
   {C2}
   cmp rax, 0
   je else2
   {e1}
   jmp end3
else2:
   {e2}
end3:
   jmp end1
else0:
   {e3}
end1:
```

How to compile relational operators




```
e1 = e2
➔
{e1}
mov [rsp - {temp_offset}], rax ; move into temporary
{e2}
cmp rax, [rsp - {temp_offset}] ; compare to temporary
mov rbx, 1
mov rax, 0
cmove rax, rbx      ; moves 1 into rax if equal
```

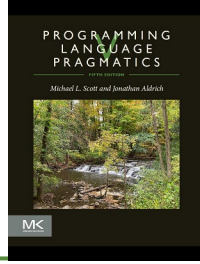
This approach is a bit wasteful when combined with if




```
   if e1 = e2 then e3 else e4
➔
   {e1}
   mov [rsp - {temp_offset}], rax
   {e2}
   cmp rax, [rsp - {temp_offset}]
   mov rbx, 1
   mov rax, 0
   cmove rax, rbx
   cmp rax, 0
   je else0
   {e3}
   jmp end1
else0:
   {e4}
end1:
```

Two comparisons, plus additional shuffling

Can we do better? Let's reconsider if



expr ::= if C then e3 else e4

➔

*compile_conditional(*C, "else0"*)*

{e3}

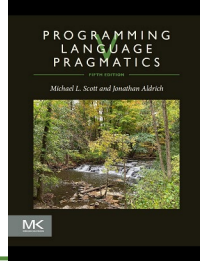
```
    jmp end1
else0:
```

{e4}

```
end1:
```

Idea: pass the else label to the conditional compilation function

Can we do better? Let's reconsider if

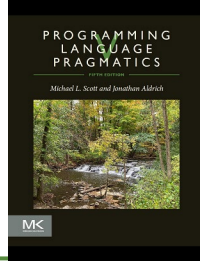


*compile_conditional(*e1 = e2, "else0")

➔

```
{e1}
mov [rsp - {temp_offset}], rax ; move into temporary
{e2}
cmp rax, [rsp - {temp_offset}] ; compare to temporary
jne else0                      ; jump to label if not equal
```

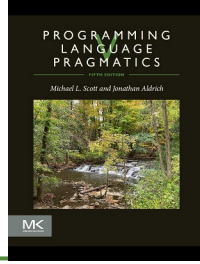
The solution is improved!




```
   if e1 = e2 then e3 else e4
➔
   {e1}
   mov [rsp - {temp_offset}], rax
   {e2}
   cmp rax, [rsp - {temp_offset}]
   mov rbx, 1
   mov rax, 0
   cmove rax, rbx
   cmp rax, 0
   je else0
   {e3}
   jmp end1
else0:
   {e4}
end1:
```

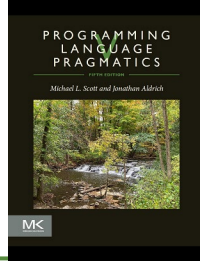
```
   if e1 = e2 then e3 else e4
➔
   {e1}
   mov [rsp - {temp_offset}], rax
   {e2}
   cmp rax, [rsp - {temp_offset}]
   jne else0
   {e3}
   jmp end1
else0:
   {e4}
end1:
```

How to compile loops




```
while C do e
→
loop0:
    compile_conditional(C, “end1”)
    {e}
    jmp loop0
end1:
```

Your turn: how to compile repeat until




```
repeat x := x+1 until x=10
```

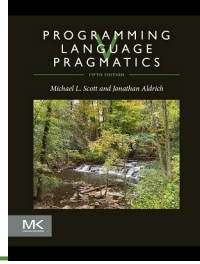
→

Note: you don't have to strictly follow the code generation approach from the last homework, just write code that works

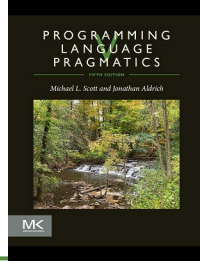
You may assume that x is stored at `x_offset` from the stack pointer `rsp`.

Enter your answer at https://forms.gle/egN39zHoWRH1zDHD6 (or the QR code above) using your Andrew ID as the email.

Don’t peek until you did the problem! Answer next…



Your turn: how to compile repeat until (SOLUTION)

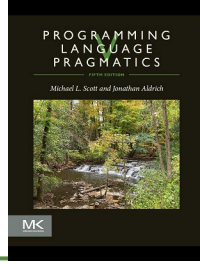



```
repeat x := x+1 until x=10
→
loop0:
    mov ax, [rsp - {x_offset}]
    add ax, 1
    mov [rsp – {x_offset}], ax
    mov ax, [rsp - {x_offset}]
    cmp ax, 10
    jne loop0
```

Your solution might differ a bit from this one

This part is basically *compile_conditional(x = 10, "Loop0")*

General approach for repeat/until




```
repeat e until C
```

→

```
loop0:
    {e}
    compile_conditional(C, “loop0”)
```